Storage Management in the Tektronix 32-Bit Smalltalk

Pat Caudill

Computer Research Laboratory
Tektronix, Inc.

ABSTRACT

This describes the planned storage management of the new Smalltalk Interpreter. This system uses no object tables but addresses objects directly. It copies referenced data to a new memory area when an area fills up. The age of the objects is used to determine the area they are to be stored in, with newer object areas being scavenged more often than older object areas. Very large objects are allocated separately and are addressed indirectly.

September 24, 1985

Storage Management in the Tektronix 32-Bit Smalltalk

Pat Caudill

Computer Research Laboratory
Tektronix, Inc.

Overview

This document is part of a preliminary design for the storage management functions in the 32-Bit Smalltalk. It is designed to supplement the pseudo-code for the storage management functions.

The large numbers of objects available in a 32-bit object number system make a large object table unwieldy. Also in a virtual memory environment there is considerable thrashing due to indirect references and especially in scanning the free list for new objects. Because of these problems the new interpreter will have its OOPs directly address objects by their virtual addresses.

~~The reference counting scheme used in previous Smalltalk interpreters has caused considerable CPU overhead. Although reference counting was used in the original Smalltalk, with several modifications to improve efficiency, it caused considerable complexity in the code.~~ The current Smalltalk uses a reference counting scheme to determine when objects could be disposed of. This caused a high overhead at execution time. Several modifications were made which brought this cost down, at the expense of code complexity. It was decide to forego the use of reference counts in this version. Instead mark/sweep like collections of memory would be used. By only sweeping those sections of memory which were likely to have large percentages of garbage the cost could be kept in line.

Experience with the current Smalltalk system has shown that most objects are very small and have lifetimes of only a few bytecodes. A large percentage of the loads and stores are in these new objects. When these objects are allocated from a free list they are scattered across the virtual address space which causes "thrashing". Searching the free list also pages memory at a high rate. The current Smalltalk allocates objects in a 32K "nursery", to increase locality of memory references. When the nursery is full objects with non-zero reference counts are moved to normal storage. Tests showed that over 98% of the objects were never moved from this nursery. This suggested that a memory management scheme which depended on the age of the objects would give better preformance.

Grades

The scheme proposed is to break the memory up into sections. Each section would hold hold objects which have been allocated for about the same amount of time. I have called these sections "Grades" after the "Nursery" they originated from. Objects are always allocated in the first grade. As this grade fills objects which survive slowly graduate to higher grades.

This keeps objects which were created together close together in memory. In particular it keep the very new objects isolated in one memory area which may be locked into real memory. When a grade runs out of memory only it is scavenged for live objects. Since higher grades will fill only as those below overflow into it, the lower grades will be garbaged collected most frequently. This allows the system to move and compact the most active objects without worrying about the many less used objects. It also compacts the used objects into a smaller amount of

virtual memory giving a greater locality of reference.

There are two memory spaces (Baker spaces)associated with each grade. The active space contains the objects currently associated with this grade. New objects placed in this grade are allocated from this space. When a grade is marked for compaction, the other space becomes important. The empty space is a block of memory with nothing in it. At scavenging time most live objects from the active space are copied into the empty space. Then the roles of the active and empty spaces are reversed. There is actually no need for the empty space as the memory could be allocated at the beginning of the garbage collection and the old active space freed at the end, but this was felt to be too much overhead.

For each object a count of the number of collection cycles in the current grade is maintained. When a critical age (set for each grade) is reached the object is not copied to the grades empty space. Instead it is promoted to the next grades active space, and its age is reset to zero.

Spaces

Each space has a discripter holding pointers to various portions of the spaces memory. There is a pointer to the bottom and top of the space to reset it and an end pointer and an alarm pointer. The end pointer points to the first nonused memory in the space. The alarm pointer is used to signal when the space is nearly full.

The memory in each space is used from both ends toward the middle. The end pointer works up from the bottom and is the address of the next object to be allocated in this space. The end is then incremented by the size of the object. When it passes the alarm pointer the grade is marked for garbage collection. There is sufficient memory for several small objects to be allocated in the space above the alarm point. This allows all garbage collections to occur between bytecode allowing some simplification in those bytecodes which allocate new objects. All the objects are allocated in a dense hunk at the bottom of the space. Since there is no need to search a free list locality of reference is preserved.

The upper portion of the space is used to hold the remembered set for this grade. This set grows down pushing the alarm point before it. More discussion of the remembered set is in section 7 Efficiency improvements.

Big Objects

Large objects are handled differently than normal objects. Larger objects tend to have a longer lifetime than small objects. This would mean that they would be copied more and would be more data to copy. They would also fill the memory space for a grade causing premature promotion of other objects. These objects are often the targets of the "becomes:" operation. Since these objects can only be addressed through indexing operations only selected bytecodes will be affected by treating them differently.

The large objects are split into a base section and the indexed portion. The base portion is a normal object. All of the fixed fields of the object are allocated with the base portion. If the indexed portion is larger than a specified size then it is allocated separately from a heap and addressed indirectly through the base portion otherwise it it allocated as an extension of the base object.

Objects which have the separated indexable fields are marked as "indirect" objects. They have a field which points to a descripter in the Big Object Table. This descripter gives the size of the indexable portion, it's address, and a back pointer to the base portion. The indexable portion is allocated from a separate heap and is never moved.

Scavenging

When a memory space is filled it signals that it's grade is to be scavenged. This consists of copying all referenced objects from that space and marking it empty. The objects are copied into two spaces. First, the empty space for this grade for objects which have been in this grade for less than the required number of scavenges. The other is the active space for next grade, for those objects which are to be graduated. For the maximum grade, these are the same. The program must check that the graduate space has not overflowed before moving a new object in. If it has, it must be marked for collection and the object "held back" for another collection. There will always be enough room for the objects in the empty space.

Where do we find the objects which are referred to from outside? There are four places. The Interpreter references some objects this includes the current method, the current context, all the semaphores associated with outside events (e.g. the keyboard or mouse) and miscellaneous others. The garbage collector must know about these (which are fixed) and handle them as special cases. The context stack is a prime source of references to objects so it must be scanned and all referenced objects copied. All objects in younger and older grades must be scanned and any they refer to in the target grade must be moved.

After all the objects which are referenced outside of the grade are copied, the transitive closure of copied objects is taken. This is done by scanning the copied objects and copying any they refer to. The locality of reference for the new space may be improved by calculating the transitive closure at intervals rather than at the end of the collection.

As each object is moved it's new address is stored in the old object and a bit is set to show this object has been forwarded. When an object is considered for moving the forwarding bit is tested and the object is only copied once. The forwarding pointer ~~can be~~ is used to update pointers to the object.

Efficiency Improvements

There are several things that can be done to speed up this process. Since most of the garbage collections are of the lower grades, if we can avoid scanning the complete upper grades this would save a lot of time. Also it is thought that most references are from newer to older objects. By keeping track of references from older objects to younger objects we can avoid scanning the objects in older grades. This will reduce the bulk of our scanning. This remembered set is kept in the grade of the referenced object. Each remembered element contains the object which references an element in this grade, and the pointer which is the actual reference. Also as a check the object which is referenced is also kept. This allows us to determine if the reference has been over written.

The remembered sets are stored at the top of the active space and push the alarm point down before them. If they cause the space to overflow they are first compacted, and then if that doesn't work the space is marked for compaction.

A large percentage of the stores are to the current context. If the remembering of stores to contexts could be deferred then all the opcodes which store to contexts would not have this overhead. This would be a big savings since most contexts are on the stack which is scanned anyway. This can be done by remembering all contexts which are activated and which are real objects. When a scavenge is made the contexts on this list are scanned and any objects they refer to in younger grades are remembered. The savings can be very large as references which have been stored over or poped off of a stack are not remembered. Note that contexts in grade 1 do not have to be scanned.

Updating

After objects are moved, all references to them must be updated. This is typically done as the references are traced. There are two groups of objects which must be updated.

The remembered sets for younger grades must be scanned for elements from the scavenged grade. When these are found the remembered set pointers are updated using the forwarding pointer. Some cleverness must be used with indirect objects as the pointer may or may not move when the object does.

The other set which must be scanned is the big object table. All references to base objects in this grade must be updated by using the forwarding pointer. If the base object was not forwarded this object is no longer referenced and the BOT entry and data area can be freed.

Code

The code which follows is the proposed memory manager written in a pseudo code like C. The code includes some English statements for things which do not have C equivalents. The size of an object is not consistent. It will probably be in bytes in the final code but often appears as the number OOPs in this code. The use is not consistent. A lot of optimization will also take place such as assigning appropriate variables to registers and expanding in place routines used only in one place.

[TABLE OF CONTENTS]

genr.c

```
1
2       typedef struct obj
3   1>      {
4           int         objsize  ;
5           int         objgrade ;
6           int         objage   ;
7           int         objhash  ;
8           int         objfixed ;
9           boolean     isForwarded ;
10          boolean     isContext   ;
11          boolean     isPointers  ;
12          boolean     isIndirect  ;
13          union
14  2>          {
15              struct obj      *class ;
16              struct obj      *forwardingPointer ;
17  2<          }
18          struct obj *objcontents[] ;
19  1<      }  object ;
20
21      typedef struct Bote             /* big object table entrys      */
22  1>      {
23          object **botref ;           /* back pointer to object       */
24          object  *botdat[] ;         /* pointer to memory space      */
25          int      botsize ;          /* memory size                  */
26  1<      }  bote ;
27
28      typedef struct
29  1>      {
30          word_t      *spcfirst ;     /* The first byte of the space. */
31          word_t      *spcend   ;     /* The last used word.          */
32          word_t      *spcmax   ;     /* The overflow point.          */
33          word_t      *spclast  ;     /* The last of the allocation.  */
34  1<      }   space ;
35
36      typedef struct
37  1>      {
38          object  *remobj ;           /* The object which is remembered       */
39          object **remptr ;           /* The down pointer in the object.      */
40          object  *remtgt ;           /* The target of the remembered pointer */
41  1<      }  remembered ;
42
43      typedef struct
44  1>      {
45          space  *active ;            /* pointer to space holding objects     */
46          space  *inactive  ;         /* pointer to space available           */
47          remembered *Remembered ;    /* pointer to remembered set            */
48          int     oldAge ;            /* cycles until objects are promoted    */
49  1<      } grade ;
```

Struct Bote *BotMAT /* Link to next in this Grade */

Bote *Bot /* Pointer to Big Objects in this Grade */

```
50
51      #define BOToffset 0
52      #define BOTpnt(exp) ((bote *)((exp)->contents[BOToffset]))
53      #define CSPoffset 3
54      #define CSPval(exp) ((int)((exp)->contents[CSPoffset]))
55      extern object **CSP
56      extern object *ContextStack[] ;
57
58      #define MAXGRADE 16
59      #define SMALLSIZE 1024
60      #define REMSIZE  100
61
62      grade school[MAXGRADE] ;
63
64      int   mustCompact  ;                    /* need to scavenge this grade */
65      int   maxAge       ;                    /* old age for current grade */
66
67      remembered *RememberedSet;              /* remembered set for current grade */
68      space      *toSpace ;                   /* space to copy to.            */
69      object     *toSpaceEnd ;                /* pointer to unused to space   */
70      space      *gradSpace ;                 /* space to age into.           */
71      object     *gradSpaceEnd ;              /* pointer to unused grad space */
72
73      #define MAXACT 50
74      #define LASTACT &(ActivatedContexts[MAXACT])
75      object *ActivatedContexts[MAXACT] ;
76  0*  object **nextActContext = { ActivatedContexts } ;
```

```
 78     /*E*/
 79     /*      salvageMemory                                   */
 80     /*   This routine is called when the memory             */
 81     /* allocator notices it is nearly out of space          */
 82     /* in the allocation area (grade-0 active).             */
 83     /* It bungles around and copies objects until           */
 84     /* either it has eliminated some, it has                */
 85     /* graduated them to an unfilled grade, or              */
 86     /* worst case it gets more memory in the oldest         */
 87     /* grade.                                               */
 88
 89     salvageMemory()
 90  1>     {
 91         int  compacting, reallocmax ;
 92
 93         if ( mustCompact < 0 ) mustCompact = 0 ;
 94         updateFromContexts() ;
 95         reallocmax = FALSE ;
 96         for ( compacting = 0 ; compacting < mustCompact ; ++compacting )
 97  2>         {
 98             compactGrade(compacting) ;
 99  2<         }
100         while ( mustCompact >= 0 )
101  2>         {
102             compacting = mustCompact ;
103             mustCompact -= 1 ;
104             compactGrade(compacting) ;
105             if ( (mustCompact == MAXGRADE) &&
106                  (compacting  == MAXGRADE) )        /* oldest filled up */
107  3>             {
108                 signal(LOWMEMORY) ;
109                 realloc ( school[MAXGRADE].inactive,
110                           school[MAXGRADE].active->spclast -
111                           school[MAXGRADE].active->spcfirst +
112                           school[MAXGRADE].active->spclast -
113                           school[MAXGRADE].active->spcend ) ;
114                 reallocmax = TRUE ;
115  3<             }
116  2<         }
117         if (reallocmax )
118             realloc ( school[MAXGRADE].inactive,
119                       school[MAXGRADE].active->spclast -
120                       school[MAXGRADE].active->spcfirst ) ;
121  1<     }
122
123
```

```
125     /*E*/
126     /*      activateContext                          */
127     /*   This routine is called to save a context    */
128     /* which is a real object when it is used as a   */
129     /* real context. References from these are not   */
130     /* marked as being from older to younger unless */
131     /* a scavenge comes up. This saves keeping track*/
132     /* of references which may (probably) will be    */
133     /* covered up. It does not save contexts in      */
134     /* grade 0 since they can't refer to younger     */
135     /* objects.                                      */
136
137     activateContext(contextOop)
138         object *contextOop ;
139 1>      {
140         if ( contextOop->objgrade == 0 ) return ;
141         *(nextActContext++) = contextOop ;
142         if ( nextActContext >= LASTACT ) /* table full so flush it. */
143             updateFromContexts() ;
144 1<      }
145
```

genr.c
updateFromContexts

```
147     /*E*/
148     /*      updateFromContexts                          */
149     /*   This routine runs through a list of the        */
150     /* contexts which are real objects and which        */
151     /* have been used since the last update and         */
152     /* inspects them for references to younger          */
153     /* objects. By doing this at the last moment        */
154     /* we dont have to check stores into contexts       */
155     /* (which are in the majority) at all. This is      */
156     /* a win since we dont have to do any checking      */
157     /* for contexts on the context stack or in          */
158     /* grade 0. Also we dont have to worry about        */
159     /* references which have been stored over.          */
160     /* The routine is a simple doublely nested          */
161     /* loop over contexts and their pointers            */
162     /* looking for pointers to younger objects.         */
163     /* Only active data in the stack part of the        */
164     /* context is actually looked at. The other         */
165     /* data is assumed not to change.                   */
166
167     updateFromContexts()
168 1>      {
169         int contextGrade, i ;
170         object    *context ;
171
172         for ( context = ActivatedContexts ; context < nextActContext ; ++context )
173 2>          {
174             contextGrade = context->objgrade ;
175             addref(context, &context->class) ;
176             for ( i = 6 ; i < CSPval(context) ; ++i )
177 3>              {
178                 addref( context, &(context->contents[i])) ;
179 3<              }
180 2<          }
181         nextActContext = ActivatedContexts ;
182         if ( current context == real object)
183 2>          {
184             activateContext(current context) ;
185             if ( current context == block context )
186                 activateContext(current context -> contents[5]) ;
187 2<          }
188 1<      }
189
```

```
233     /*E*/
234     /*      compactGrade                              */
235     /*   This routine moves all the currently         */
236     /* active objects in grade to a new space.        */
237     /* it then makes the currently active space the   */
238     /* empty one for this grade. This has the effect  */
239     /* of throwing away all the unused objects.       */
240     /* The routine first finds the spaces to which    */
241     /* objects are to be copied. This will depend on  */
242     /* whether this is the oldest grade. It then      */
243     /*  copies all stuff refered to from the context  */
244     /* stack. Next it copies all the stuff refered    */
245     /* to by older objects. Then it loops alternatly  */
246     /* copying all objects refered to by the stuff    */
247     /* moved to the "to" and "grad" spaces. (if       */
248     /* these are the same this is half a loop.)       */
249     /* Finally it updates all pointers in younger     */
250     /* objects that refered to objects in this        */
251     /* grade. Oh Yes, It swaps the active and empty   */
252     /* spaces for this grade.                         */
253
254     compactGrade(grade)
255         int grade ;
256  1>     {
257         grade       *room ;
258         int          i ;
259
260         room = &school[grade] ;
261         if ( room->active->spcend < room->active->spcmax ) return ;
262         RememberedSet = room->Remembered ;
263
264         toSpace = room->empty ;
265         room->Remembered = toSpace->spclast ;
266         room->Remembered->remobj =                  /* mark end of    */
267         room->Remembered->remptr =                  /* remembered set */
268         room->Remembered->remtgt = 0 ;
269         toSpace->spcmax = toSpace->spclast - 5*SMALLSIZE ;
270         if ( grade == MAXGRADE )
271  2>         {
272             gradSpace = toSpace ;
273  2<         }
274         else
275  2>         {
276             gradSpace = school[grade+1].active ;
277  2<         }
278         maxAge = room->oldAge ;
279         toSpaceEnd = toSpace->spcend ;              /* mark start of where  */
280         gradSpaceEnd = gradSpace->spcend ;          /* copied into.         */
281
282         scavengeInterpObjects(grade) ;              /* move misc. obj.      */
283         scavengeSpaceStartingAt(grade, toSpace, toSpaceEnd ) ;
284         toSpaceEnd = toSpace->spcend ;
285         scavengeRememberedObjects(grade, RememberedSet) ;
286         toSpaceEnd = toSpace->spcend ;
287         scavengeContextStackObjects(grade) ;
288         scavengeSpaceStartingAt(grade, toSpace, toSpaceEnd ) ;
289         for ( i = grade-1 ; i >= 0 ; --i )
290  2>         {
291             scavengeSpaceStartingAt(grade, school[i].active,
292                                     school[i].active->spcend) ;
293             scavengeSpaceStartingAt(grade, toSpace, toSpaceEnd ) ;
294             toSpaceEnd = toSpace->spcend ;
295  2<         }
296  2>     do  {
297             scavengeSpaceStartingAt(grade, toSpace, toSpaceEnd ) ;
298             toSpaceEnd = toSpace->spcend ;
299             if ( gradSpace != toSpace )
300  3>             {
301                 scavengeSpaceStartingAt(grade, gradSpace, gradSpaceEnd ) ;
```

```
302                 gradSpaceEnd = gradSpace->spcend ;
303  3<             }
304  2<         } while ( toSpaceEnd != toSpace->spcend ) ;
305
306         updateBot(grade) ;
307         for ( i = grade-1 ; i >= 0 ; --i )
308  2>         {
309             updateRemembered(school[i].Remembered ) ;
310  2<         }
311
312         room->active->spcend = room->active->spcfirst ;
313         room->empty = room->active ;
314         room->active = toSpace ;
315         return ;
316  1<     }
317
```

```
319     /*E*/
320     /*      scavengeInterpObjects                              */
321     /*   This routine checks any object the interpreter        */
322     /* knows about and scavenges them as necessary.            */
323     /* this includes the display bitmap, known objects like    */
324     /* nil, true, ..., active semaphores, and objects in       */
325     /* registers.                                              */
326
327     scavengeInterpObjects(grade)
328         int grade ;
329 1>      {
330
331         /* check any interpreter saved values here */
332
333 1<      }
334
```

genr.c
scavengeContextStackObjects

```
336     /*E*/
337     /*      scavengeContextStackObjects                     */
338     /*   This routine scans the context stack and copies    */
339     /* any object from the target grade which is referenced */
340     /* to the correct receiving space. The context stack    */
341     /* reference is updated.                                */
342
343     scavengeContextStackObjects(grade)
344         int     grade ;
345 1>      {
346         object **cip ;
347
348         for ( cip = CSP ; cip > ContextStack ; --cip )
349 2>          {
350             if ( (**cip)->objgrade == grade )
351 3>              {
352                 copyAndForwardObject(*cip) ;
353                 *cip = **cip->forwardingPointer ;
354 3<              }
355 2<          }
356 1<      }
357
```

```
359     /*E*/
360     /*      scavengeRememberedObjects                          */
361     /*   This routine scans all the objects in a remembered */
362     /* set and copys any in the specified grade. It also    */
363     /* updates the references to those objects. It checks   */
364     /* the target to see that it hasn't already been moved  */
365     /* and hasnt been superseeded. The end of the remembered*/
366     /* set is marked by a zero entry.                       */
367
368     scavengeRememberedObjects( memorys)
369         remembered *memorys ;
370 1>      {
371         remembered *rip ;
372
373         for ( rip = memorys ; *rip ; ++rip )
374 2>          {
375             copyRememberedObject(rip) ;
376 2<          }
377 1<      }
378
```

```
380     /*E*/
381     /*      copyRememberedObject                                */
382     /*   This routine copys an object which is in a             */
383     /* remembered set to the tospace or gradSpace as approp.*/
384     /* It doesn't copy an object if it has already been        */
385     /* moved or if the reference to it has been changed.       */
386     /* It doesnt update the remembered set for the new         */
387     /* space since that should happen when that space is       */
388     /* scaned later.                                           */
389
390     copyRememberedObject(remref)
391         remembered *remref ;
392 1>      {
393         object  *reference ;
394
395         reference = *(remref->remptr) ;
396
397         if ( isInt(*reference) ) return ;
398         if ( *reference->isForwarded) return ;
399         if ( remref->remtgt != reference) return ;
400
401         copyAndForwardObject(reference) ;
402         return ;
403 1<      }
404
405
```

genr.c

scavengeThisSpaceStartingAt

```
407     /*E*/
408     /*      scavengeThisSpaceStartingAt                         */
409     /*   This routine does a scan of the objects that have      */
410     /* been moved starting at currentObject. It copies any      */
411     /* un-copied objects from this grade that they refer to.*/
412
413
414     scavengeSpaceStartingAt(grade, theSpace, currentObject)
415                     int     grade ;
416                     space  *theSpace ;
417         object *currentObject ;
418  1>     {
419
420         for(  ;
421               currentObject < theSpace->spcend ;
422               currentObject += ((currentObject->objsize + 3) & ~3) )
423  2>           {
424               scavengeReferentsOf(grade, currentObject ) ;
425  2<           }
426  1<     }
427
```

```
429     /*E*/
430     /*      scavengeReferentsOf                                 */
431     /*   This routine scans an object and promotes any          */
432     /* objects at which it points that are in grade "grade" */
433     /* It always copies the class, then if the object is    */
434     /* not pointers it quits. Otherwise it copies all stuff */
435     /* pointed in the contents of the object to the end or  */
436     /* for contexts through the stack pointer.              */
437
438     scavengeReferentsOf(grade, anObject )
439                 int     grade ;
440         object *anObject ;
441 1>      {
442         object **start ;
443                 object **end ;
444
445         if ( anObject->objsize == 1 ) return ;
446
447         if ( anObject->class->objgrade == grade)
448 2>          {
449             copyAndForwardObject(anObject->class) ;
450             anObject->class = anObject->class->forwardingPointer ;
451             addref(anObject, &(anObject->class) ;
452 2<          }
453
454         if ( !anObject->isPointers) return ;
455
456         start = &anObject->contents[1] ;
457         if ( anObject->isContext )
458 2>          {
459             end = &(anObject->contents[CSPval(anObject)]) ;
460 2<          }
461         else if ( anObject->isIndirect )
462 2>          {
463             end = &anObject->contents[anObject->objsize-1] ;
464 2<          }
465         else
466 2>          {
467             end = &anObject->contents[anObject->objsize] ;
468 2<          }
469
470         while ( start <= end )
471 2>          {
472             if ((*start)->objgrade == grade )
473 3>              {
474                 copyAndForwardObject((*start)) ;
475                 *start = (*start)->forwardingPointer ;
476                 addref(anObject, start ) ;
477 3<              }
478 2<          }
479
480         if ( anObject->isIndirect )
481 2>          {
482             start = BOTpnt(anObject)->botdat ;
483             end = start + (BOTpnt(anObject)->botsize ) ;
484 2<          }
485
486         while ( start <= end )
487 2>          {
488             if ((*start)->objgrade == grade )
489 3>              {
490                 copyAndForwardObject((*start)) ;
491                 *start = (*start)->forwardingPointer ;
492                 addref(anObject, start ) ;
493 3<              }
494 2<          }
495         return ;
496 1<      }
497
```

```
191     /*E*/
192     /*      addref                                  */
193     /*   This routine checks the reference at refptr*/
194     /* in object objptr for being a pointer from    */
195     /* older to younger objects. If it is then the  */
196     /* reference (and its object) are saved in the  */
197     /* remembered set for the grade of the pointee  */
198     /* object. The target object is also saved for  */
199     /* checking that the value has not been replaced*/
200     /* If the memory space overflows then a garbage */
201     /* collection is signaled.                      */
202
203     #define addref( obj, ref ) if ((obj)->objgrade > (*(ref))->objgrade) \
204                                    addrefS(obj, ref ) ;
205     addrefS(objptr, refptr)
206         object  *objptr ;           /* object to be remembered             */
207         object **refptr ;           /* pointer in object pointing back     */
208 1>      {
209         grade *room ;               /* room containing pointee object      */
210         object *target ;            /* pointee object.                     */
211         remembered *prset ;         /* entry for remembering in            */
212         space *daspace ;            /* the space which contains the target. */
213
214         target  = *refptr ;
215         room    = &school[target->objgrade] ;
216         prset   = room->Remembered -= sizeof(remembered) ;
217         daspace = room->active ;
218
219         prset->remobj = objptr ;
220         prset->remptr = refptr ;
221         prset->remtgt = target ;
222
223         daspace->spcmax  -= sizeof(remembered) ;
224         if ( daspace->spcend > daspace->spcmax )
225 2>          {
226             mustCompact = target->objgrade ;
227             signal(CompactMemory) ;
228 2<          }
229         return ;
230 1<      }
231
```

```
if (daspace->spcend > daspace->spcmax)
{
Room->Remembered = compactRemembered(Set
    (Room->Remembered, daspace->spclast);
Spcmax = Room->Remembered - smallsizeof;
}
```